Sistemi Operativi, Secondo Modulo, Canale M–Z
Riassunto della lezione del 03/04/2017

Igor Melatti

Espansioni nella bash

- Arithmetic expansion
  - tutte le operazioni sugli *interi*
  - vengono valutate, e il risultato viene sostituito all'interno del comando
  - sintassi: `$((expr))`, dove expr è scritta come una normale espressione algebrica (anche con le parentesi); con `%` che è il modulo
  - si possono fare anche confronti (ma stavolta 0 è falso, diverso da 0 è vero)
  - anche operazioni bit-a-bit, come nel C, e operazioni logiche (OR, AND, NOT; per il bit-a-bit c'è anche lo XOR)
  - si possono usare anche le variabili (quelle locali o d'ambiente della bash), tanto la variable expansion avviene prima... in più, si possono riferire le variabili anche senza `$`
  - si possono fare assegnamenti, anche con operatori di modifica (del tipo `+=`), e di incremento/decremento (`++`, `--`), sia pre che post
  - esiste l'espressione condizionale con il `?`
  - è possibile fare assegnamenti e poi dare un'espressione da valutare: `,`
  - se viene usata una variabile che non è stata definita, viene espansa a 0
  - se una variabile dentro un'espressione aritmetica espande come una stringa, l'espansione prova ad espandere quella stringa come se fosse una variabile (il processo è ricorsivo; se si arriva ad una variabile non definita, il suo valore è 0; se si arriva ad una variabile con un nome non valido, dà errore)!
  - per inciso: la sintassi `(())` può essere usata anche senza $: è un comando che può essere usato per assegnare valori

  - **esercizio:** verificare se la lunghezza di `PATH` è maggiore della lunghezza di `PWD` oppure la vecchia current directory ha più caratteri di quella attuale; dipendentemente da questo, assegnare alla variabile `var` il valore 10 (se è vero) o 20 (se è falso)
  - **esercizio:** verificare se il valore di `BASH_PID` è pari o dispari

- Command substitution
  - due sintassi: `` `comando` `` (backticks) e `$(comando)`
  - si possono fare annidamenti (nel primo caso, occorrerà usare \`)
  - l'espansione consiste nell'eseguire il comando, prendere ciò che viene scritto in stdout e sostiturlo al comando stesso
  - **esercizio:** come si può, con un solo comando, cercare dei file che corrispondono ad un certo pattern, e poi cercare nei files di tale insieme una certa stringa? Farlo in due modi: i) usando una variabile d'appoggio e la concatenazione tra comandi; ii) usando un comando solo e la command substitution
  - **esercizio:** creare un array dove sono definiti i seguenti indici: `3`, `5a`, `10df`, `100`, con valore pari al doppio della parte numerica dell'indice; assegnare ad una variabile `var` il numero di indici assegnati in tal modo

- Word splitting: attenzione, si applica solo al risultato di parameter, arithmetic e command expansion, e solo se non double quoted (se fosse single quoted, niente espansione...)
  - semplicemente, ogni diversa parola è passata separatamente al comando
  - normalmente, le parole sono separate da spazi o tab
  - nel caso della bash, sono in realtà separate dai caratteri dentro la variabile d'ambiente `IFS`
  - esempio: supponiamo di settare `IFS=:`
  - il comando `echo a:b` scrive `a:b`
  - il comando `echo "a:b"` scrive `a:b`
  - ma questo solo perché il word splitting qui non avviene, in quanto non c'è stata nessuna delle 3 espansioni (di variabili, aritmetica o di comando) richieste
  - infatti, il comando `` echo `echo a:b` `` scrive `a b`
  - mentre il comando `` echo "`echo a:b`" `` scrive `a:b` (dentro i double quotes, niente word splitting)
  - un risultato analogo lo si può ottenere con il comando `var="a:b"; echo $var; echo "$var"` (dove si usa l'espansione dei parametri anziché la command substitution)

- analogamente, `echo $IFS` non scrive nulla: infatti dapprima la bash espande `$IFS` in `:`, e poi viene fatto il word splitting, che sostituisce il `:` con uno spazio
- invece, `echo "$IFS"` scrive `:`, perché il word splitting non avviene
- si può ora capire la differenza tra `$*` e `$@`: nel primo caso, vengono scritti tutti gli argomenti separati dal primo carattere di `IFS`, nel secondo invece vengono scritti tutti gli argomenti separati da spazi (indipendentemente dal valore di `IFS`)
- se non sono double quoted, il risultato finale è quindi lo stesso: `$*` va soggetto a word splitting e il carattere `IFS` viene sostituito da spazi; `$@` va anch'esso soggetto a word splitting, ma non c'è nulla da sostituire, in quanto sono già tutti spazi
- se sono double quoted: non andando soggetto a word splitting, `$*` rimarrà separato dal carattere `IFS`, mentre `$@` rimane separato da spazi
- quindi, nel caso siano double quoted, il risultato sarà diverso non appena `IFS` non contenga lo spazio
- una precisazione importante: diversamente da altre variabili d'ambiente, `IFS` viene sempre resettata al suo valore di default (che è `$' \t\n'`) quando viene invocato uno script
- esempio: sia `var.sh` il seguente script:

```
echo $HOME
echo $PATH
echo $PS4
echo "$IFS"
```

- dopo averlo reso eseguibile, provare ad invocarlo, per esempio, così: `HOME="ciao" PATH="aho" PS4="ehi" IFS=";" ./vars.sh`
- l'unica variabile non settata come indicato è proprio `IFS`; notare che vengono scritte 2 andate a capo: una è dovuta ad `IFS` stesso, l'altra ad `echo`
- **esercizio:** nella stampa ottenuta con il comando precedente, in realtà la quarta riga contiene prima uno spazio, poi un tab e poi un'andata a capo: trovare il modo per provare questa affermazione (senza usare il mouse...)
- per provarlo, scrivere dentro un file `chiocc.sh` la riga `IFS=";"; echo $@; echo "$@"`, e dentro un file `ast.sh` la riga `IFS=";"; echo $*; echo "$*"`
- supponendo che `IFS` valga `":"`, provare a chiamare `bash chiocc.sh a:b` e confrontare con `bash ast.sh a:b`
- il risultato è lo stesso: il word splitting non avviene, quindi ai due script viene passato `a:b`, che è un argomento unico; pertanto, né `$*` e `$@` hanno necessità di mettere separatori

  - provare a chiamare `bash chiocc.sh $(echo a:b)` e confrontare con `bash ast.sh $(echo a:b)`
  - il risultato sarà diverso nella seconda linea, come predetto; notare che il carattere usato per separare il risultato di `"$*"` è l'IFS settato *dentro* lo script
  - infatti, il word splitting fa sì che i due script vengano chiamati con `a b`, quindi con 2 argomenti; dopodiché, il discorso fatto sopra fa il resto
  - supponendo che `IFS` contenga `:`, provare a chiamare `bash chiocc.sh "$(echo a:b)"` e confrontare con `bash ast.sh "$(echo a:b)"`
  - il risultato sarà uguale: non c'è word splitting nel passare gli argomenti, quindi ai due script viene passato `a:b`, che è un argomento unico
  - **esercizio:** si supponga di voler vedere le informazioni standard sulle directory contenuto in `PATH`, settando opportunamente `IFS`. Funziona il comando `ls -ld $PATH`? Perché? E il comando `ls -ld "$PATH"`? Perché? E il comando `ls -ld $(echo $PATH)`? Perché? Come si può correggere, minimalmente, quest'ultimo comando?
  - **esercizio:** Si supponga che il contenuto di `PATH` venga messo su un file. Come si può ottenere lo stesso risultato dell'esercizio precedente?
  - **esercizio:** confrontare il risultato dei comandi `echo ‘ls‘` e `echo "‘ls‘"`. Cosa succede, e come si spiega?

- Filename expansion (o globbing); `man 7 glob`
  - vedere lezione 5
  - **esercizio:** qual è il risultato del comando `ls index.{html,p*}`?
  - Creare 2 file in una directory inizialmente vuota: `filename.txt` e `file name.txt`. Che differenza c'è tra i comandi `ls *`, `ls ‘ls‘`, `ls "*"`, `ls "‘ls‘"`?

- Quote removal
  - alla fine di tutte le espansioni, una diminuzione: tutte le occorrenze dei caratteri \, `"` e `’` vengono eliminate
  - a meno che non siano state prodotte da una qualche expansion: `echo ‘echo \"‘`

Gli Script Bash

- Come detto, sono file di testo che contengono comandi bash
  - gli stessi che possono essere scritti anche sulla shell interattiva

  - il comando `exit n` fa sì che l'exit code dell'esecuzione di uno script sia `n` (tra 0 e 255)
  - i commenti si fanno con il `#`, analogamente al `//` del Java (quindi, da dove si trova fino all'andata a capo)
  - eccezione 1: nell'espansione di parametri (vedere lezione 10)
  - eccezione 2 (*shabang*): nella prima riga dello script, se seguito dal carattere `!`, ciò che segue viene interpretato come il programma (l'interprete, da fornire col percorso *assoluto*) tramite il quale lanciare lo script stesso (questo solo se lo si rende eseguibile e lo si lancia direttamente)
  - in assenza di tale riga, si usa la bash stessa
  - lo si può usare per rendere eseguibile qualsiasi file di testo per qualsiasi interprete: ad esempio, Python, Perl, ...
  - non Java, perché non prende direttamente in input il file da eseguire
  - per awk, dato che per passargli un file con il programma occorre l'opzione `-f`, occorre scrivere `#!/usr/bin/awk -f`
  - `https://it.wikipedia.org/wiki/Shabang`
  - **esercizio:** trasformare tutti gli esercizi su `awk` e `sed` nelle lezioni 7 ed 8 in script eseguibili; si può fare anche con `uniq`?

- Comandi built-in e di sistema: distinguibili o dal `man` o più semplicemente con il comando (built-in!) `type`
  - provare ad esempio `type cd` e `type which`

- Come detto in lezione 2, la bash ha la `history`
  - interattivamente, la si può sfruttare con le frecce posizione su/giù, o con il CTRL+r per cercare una sottostringa
  - una volta trovato un comando precedente, lo si può modificare
  - tutto ciò lo si può fare anche da comando
  - *history expansion*: ovviamente, avviene prima di ogni altra expansion
  - `!n` viene espanso nell'`n`-esimo comando dall'inizio della shell (se `n` è positivo), o nell'`n`-esimo comando dato in precedenza (se negativo)
  - `!0` dà errore; `!-1` si può scrivere anche `!!`
  - in realtà, la history della bash ha un limite, quindi `!1` non è necessariamente il primo comando dato...
  - comando (built-in) `history`, mostra tutti i comandi nella history
  - `!prefix` trova ed esegue l'ultimo comando che iniziava con `prefix` (senza spazi...)

  - `!!:s/search/replace/` se l'ultimo comando che conteneva `search`, prima di eseguirlo sostituisci `search` con `replace` (si può fare la stessa cosa con `^search^replace ^`)
  - si può fare molto altro; come al solito, vedere `man bash`
  - **esercizio:** farsi stampare su stdout (senza eseguirli): i) il primo e l'ultimo comando della history; ii) il comando a metà della history, ma se contiene il comando `ls` sostituirlo con `stat`

- Le condizioni negli script (e nella bash interattiva)
  - sono usate nelle istruzioni condizionali `if-then` ed `if-then-else`, nonché nei cicli `while` ed `until`
  - si possono scrivere in due modi (quattro contandone anche altri 2 non standard): `[ condizione ]` (occhio agli spazi: uno dopo la parentesi aperta e uno prima di quella chiusa) e `test condizione`
  - si possono fare anche combinazioni logiche di condizioni: `-a` per l'AND, `-o` per l'OR, `!` per il NOT; per raggruppare, `\(` e `\)`
  - tutte le principali condizioni sono riportate in Tabella 1
  - formalmente, viene avviato un processo che effettua il controllo della condizione, e poi ritorna come exit status 0 (condizione vera) o 1 (condizione falsa); quindi si possono testare anche con `echo $?`
  - sintassi alternativa 1: `[[`
    * rispetto a `[`, cambiano alcune cosette: le combinazioni logiche sono come nel C (quindi con `&&`, `||` e `!`); i confronti si fanno con `<` e `>`, e si usa `==` anche tra interi
    * ammette anche confronti con espressioni regolari: `[[ expr =~ regex ]]` (sono quelle estese, ma senza backreference)
  - sintassi alternativa 2: `((`, vale solo per gli operatori aritmetici
  - **esercizio:** Farsi stampare 1 se esistono due file di nome `file1.txt` e `file2.txt`, con il secondo più recente del primo, oppure se `file1.txt` è un link simbolico; altrimenti, farsi stampare 10 (usare solo comandi e/o espansioni visti fino a questo punto).

- Esecuzione condizionale: `if test-command; then commands1; else commands2; fi`
  - l'`else` può non esserci
  - o può essere sostituito da un `elif`
  - qui come altrove: occhio ai punti e virgola, vanno esattamente lì
  - ogni punto e virgola può essere sostituito da un'andata a capo (nella shell interattiva, verrà atteso che si completi il comando)
  - si può andare a capo anche in alcuni punti dove non c'è il punto e virgola (ad esempio, dopo le keyword)

Table 1: Condizioni in bash

Operatore	**Operandi**	**Vero se...**	**Binario**
`-d p`	pathname	p è una directory	unario
`-e p`	pathname	p esiste	unario
`-f p`	pathname	p è un file regolare	unario
`-h p`	pathname	p è un link simbolico	unario
`-s p`	pathname	p ha dimensione non nulla	unario
`p1 -nt p2`	pathname	p1 è più recente di p2	binario
`p1 -ot p2`	pathname	p2 è più recente di p1	binario
`s1 == s2`	string	s1 ed s2 sono uguali	binario
`s1 != s2`	string	s1 ed s2 sono diverse	binario
`s1 \> s2`	string	s1 è lessicograficamente maggiore di s2	binario
`s1 \< s2`	string	s1 è lessicograficamente minore di s2	binario
`-z s`	string	s ha lunghezza 0	unario
`-n s`	string	s ha lunghezza maggiore di 0; `-n` può anche essere omesso	unario
`i1 -eq i2`	integer	i1 è uguale ad i2	binario
`i1 -ne i2`	integer	i1 è diverso da i2	binario
`i1 -gt i2`	integer	i1 è maggiore di i2	binario
`i1 -lt i2`	integer	i1 è minore di i2	binario
`i1 -ge i2`	integer	i1 è maggiore o uguale a i2	binario
`i1 -le i2`	integer	i1 è minore o uguale a i2	binario

  - `commands` è un qualsiasi comando della shell (quindi anche una sequenza, condizionale o no, di comandi)
  - l'exit status dell'intero comando è 0 se nessun `test-command` è risultato vero, altrimenti è l'exit status dell'ultimo `commands` eseguito
  - **esercizio:** rifare l'esercizio precedente usando un `if`

- Esecuzione condizionale: `case word in listapattern1) commands1;; ...listapatternn) commandsn;; esac`
  - i pattern sono quelli del filename expansion
  - ci possono essere liste di pattern, separati da `|`
  - si esegue il comando del primo pattern che fa match
  - ciò permette di mettere come ultimo pattern `*`, ad indicare "qualsiasi altra cosa"
  - nota: i pattern non vanno quotati (né single né double), altrimenti hanno il loro valore letterale
  - **esercizio:** scrivere uno script che accetti 2 argomenti e 3 opzioni. Come di consueto, si supponga che le opzioni, introdotte da `-a`, `-b` e `-c`, vengano prima degli argomenti. Le opzioni `-a` e `-b` necessitano un argomento (per esempio, come l'opzione `-o` di `ps`), mentre `-c` è solo un flag (senza argomenti, come ad esempio l'opzione `-E` di `grep`). Come risultato, lo script deve avere il seguente output:

    ```
    Opzione -a: A
    Opzione -b: B
    Opzione -c: C
    Argomento 1: A1
    Argomento 2: A2
    ```

    dove A, B e C possono essere o `assente` (se l'opzione non è stata data) oppure il valore passato all'opzione stessa (nel caso di C, dev'essere semplicemente `presente`). Se viene data un'opzione non tra quelle elencate, oppure gli argomenti non sono esattamente 2, occorre dare un messaggio d'errore. Provare a fare questo script sia usando quanto visto sin qui, sia usando il comando built-in `getopts` (vedere il `man bash`)

- Ciclo `until test-command; do commands; done`
  - `commands` viene eseguito tutte le volte che `test-command` fallisce (ovvero, torna qualcosa di diverso da 0), finché il test non esce con 0 (successo)
  - l'exit status dell'intero comando è 0 se nessun `commands` è stato eseguito, altrimenti è l'exit status dell'ultimo `commands` eseguito

  - **esercizio:** Scrivere uno script che prende 2 argomenti e copia il primo nel secondo. La copia deve avvenire solo se: i) il primo file esiste ed è accessibile in lettura; ii) il secondo file è un file con i permessi di scrittura oppure il secondo file è una directory con i permessi di scrittura. La copia deve avvenire tramite un ciclo `until` in cui la copia avviene al ritmo di una riga per iterazione, fino al completamento della copia. L'exit status dello script deve coincidere con quello del ciclo `until`. Verificare tale exit status nel caso in cui il primo file sia vuoto. Modificare poi lo script in modo da prendere un'opzione `-r` r (se non dato, $r = 1$): il ritmo di copiatura sarà ad r righe per iterazione.

- Ciclo `while test-command; do commands; done`
  - come sopra, ma questa volta `commands` è eseguito se il test ha successo, e si esce dal ciclo al primo fallimento
  - **esercizio:** rifare l'esercizio precedente, aggiungendo una quarta opzione (flag) `-w`. Se `-w` viene dato, allora dev'essere usato un ciclo `while`, altrimenti usare il ciclo `until` come sopra.

- Ciclo `for`, 2 sintassi:
  1. `for name [in listaparole]; do commands; done`
     - le parole nella lista di parole sono sempre separate da spazi, ma la lista stessa può essere il risultato di un word splitting
     - se la lista di parole non c'è, si intende essere `"$@"`
     - viene eseguito `command` tante volte quante sono le parole in `listaparole` (attenzione al word splitting...); ogni volta, prima di eseguire `command`, alla variabile `name` viene assegnata una parola dalla lista, nell'ordine specificato
     - exit status: quello dell'ultimo comando in `commands`, oppure 0 se `listaparole` viene espansa alla lista vuota
     - **esercizio:** eseguire `stat` su ogni directory elencata in `PATH`.
  2. `for ((expr1; expr2; expr3)); do commands; done`
     - praticamente, come il C (e il Java)
     - `expr1` è l'inizializzazione (eseguita una sola volta, prima di eseguire `commands` la prima volta)
     - `expr2` è la condizione da controllare prima di ogni esecuzione di `commands`
     - `expr3` è il comando da effettuare dopo ogni esecuzione di `commands`
     - in queste 3 espressioni, vale quanto detto per le espressioni aritmetiche (vedere lezione 13)
     - exit status: quello dell'ultimo commands, oppure 1 se c'è qualche errore in una delle 3 espressioni

  - **esercizio:** rifare l'esercizio precedente, cambiando la quarta opzione in `-w` *w*. Se *w* è `while`, allora dev'essere usato un ciclo `while`, se è `until` usare il ciclo `until`, se è `for1` usare il ciclo `for` nella prima forma (difficile...), se è `for2` usare il ciclo `for` nella seconda forma, altrimenti dare un messaggio d'errore e non fare nulla.

- Le cattive abitudini restano: `break` e `continue`
  - `break` forza l'uscita da un ciclo
  - `continue` forza la continuazione di un ciclo (nel caso del `for` con 3 espressioni, viene prima eseguita `expr3`)

- Script interattivo
  - un po' ossimorico, ma c'è il comando `read` {`nomevar`}
  - mette in `nomevar` la stringa letta da tastiera (o meglio, da stdin)
  - se ci sono più variabili date come argomento, allora assegna ciascuna diversa parola ad una variabile
  - le parole sono separate da `IFS`
  - **esercizio:** Scrivere uno script che, senza usare `awk` o `sed`, legga un file (di testo) riga per riga; per ogni riga letta, sostituisca la parola `ciao` con `ehi`, e ogni numero con il suo successivo.